Bab 3. OBJECT ORIENTED DESIGN

OBJEKTIF:

1. Mahasiswa memahami pemrograman beriorientasi *object*.
2. Mahasiswa mampu memahami *static class member*.
3. Mahasiswa mampu mengidentifikasi hubungan antar kelas.
4. Mahasiswa mampu mengidentifikasi method.
5. Mahasiswa memahami konsep enkapsulasi.
6. Mahasiswa memahami interfaces.

3.1 *Object Oriented Programming*

Pada pemrograman berorientasi *object*, program dibangun dengan membuat beberapa *object* dan melakukan interaksi antara *object* tersebut. *Object* dalam pemrograman adalah komponen program yang memiliki dua kemampuan:

- *Object* dapat menyimpan data. Data yang disimpan dalam *object* disebut sebagai ***field***.
- *Object* dapat melakukan aksi atau operasi-operasi. Aksi-aksi atau operasi-operasi ini disebut sebagi ***method***.

3.1.1 *Class*

Class merepresentasikan sebuah ADT (*Abstract Data Type*/Tipe Data Abstrak). *Class* berperan sebagai sebuah *template* yang bisa kita gunakan untuk membuat banyak *object* (instansiasi). Sebuah deklarasi *class* hanya membuat *template*, tetapi tidak membuat sebuah *object*. Sebuah *class* membuat tipe data baru yang bisa digunakan untuk membuat *object*. Dengan demikian, sebuah *class* membuat logika *framework* yang mendefinisikan hubungan antara anggotanya. Ketika mendeklarasikan sebuah *object* dari *class*, kita membuat sebuah instansiasi dari *class* tersebut.

Sebuah *class* tidak hanya dapat digunakan untuk membuat sebuah *object*, namun juga dapat membuat berapapun *object* dari suatu *class*. Setiap *object* (*instance*) dari *class* memiliki nilai-nilai *attribute* tersendiri. *Attribute* dari *object* ini adalah data yang dimiliki *object*. Data-data ini disebut juga sebagai ***field***. Sedangkan *behavior* dari *object* adalah hal-hal yang dapat dikerjakan *object*. *Behavior* ini disebut sebagai ***method***.

Berikut adalah "kerangka" umum dari sebuah *class*:

```
public class Segitiga
{

}
```

Keyword `public` yang dituliskan di awal dari *header class* adalah *access specifier*. Setelah *keyword* `public`, selanjutnya adalah menuliskan *keyword* `class` yang diikuti dengan `Segitiga`, yang merupakan nama dari *class*. Selanjutnya untuk isi dari *class*, yang terdiri dari *field* dan *method*, akan dituliskan di dalam tanda kurung kurawal seperti berikut:

```
public class Segitiga
{
    private double alas;
    private double tinggi;
}
```

Class `Segitiga` mempunyai dua *field*, `alas` dan `tinggi`. *Field* adalah variabel yang digunakan untuk menyimpan data dari *object*.

Tabel berikut menjelaskan perbedaan *access specifier* `private` dan `public`:

Access Specifier	Keterangan
`private`	Jika *access specifier* `private` digunakan ke member dari *class*, member tersebut tidak dapat diakses oleh kode di luar *class*. Member ini hanya dapat diakses oleh *method* yang merupakan member dari *class* yang sama.
`public`	Jika *access specifier* `public` digunakan ke sebuah member dari *class*, member tersebut dapat diakses oleh kode di dalam *class* dan juga oleh kode di luar *class*.

Sebagai contoh, kita akan menuliskan *method* dari *class* `A`. *Method* dari *class* `A` terdiri dari `setData()` dan `display()`. Kode lengkap dari *class* `A` dapat dilihat pada kode berikut.

Definisi Class (A.java)

```
/*
     Class A
*/
public class A
{
    private int x;
    private int y;

    /*
        Method setData menyimpan sebuah nilai
        dalam field x1 dan y1.
        @param x1, y1 Nilai yang disimpan dalam field x1 dan y1.
    */
    public void setData(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    /*
        Method display()
        Menampilkan nilai field x dan y
    */
    public void display()
    {
        System.out.println(x + "\t" + y);
    }
}
```

Perhatikan pada kode *class* `A`, kita tidak menuliskan *method* `main`. Ini karena *class* bukanlah program komplit, tetapi merupakan sebuah definisi *class* `A`. Program yang membuat dan menggunakan *object* `A` akan mempunyai *method* `main` sendiri. Program berikut mendemonstrasikan penggunaan *class* `A` untuk membuat *object* `A`.

Program (ClassTest.java)

```
/*
    Program berikut mendemonstrasikan penggunaan
    method setData dan display dari class A
*/
public class ClassTest
{
    public static void main(String[] args)
    {
        // Buat sebuah object A dan
        // tugaskan alamatnya ke variabel a1.
        A a1 = new A();

        // Buat sebuah object A dan
        // tugaskan alamatnya ke variabel a2.
        A a2 = new A();

        /*
         Panggil method setData, berikan nilai
         10 dan 20 sebagai argumen.

         Menugaskan field x dengan nilai 10 dan
         field y dengan nilai 20
        */
        a1.setData(10, 20);

        /*
         Panggil method setData, berikan nilai
         5 dan 7 sebagai argumen.

         Menugaskan field x dengan nilai 5 dan
         field y dengan nilai 7
        */
        a2.setData(5, 7);

        // Tampilkan panjang dan lebar dari object PersegiPanjang
        a1.display();
        a2.display();
    }
}
```

Program `ClassTest.java` harus disimpan di dalam *folder* yang sama dengan *file* `A.java`. Perintah berikut dapat digunakan untuk mengkompilasi program:

```
javac ClassTest
```

Ketika *compiler* membaca *source code* `ClassTest.java` dan melihat bahwa *class* bernama `A` digunakan pada program tersebut, *compiler* akan mencari *file* `A.class` di dalam *folder* tempat *source code* `ClassTest.java`. Dan karena kita belum mengkompilasi `A.java`, maka tidak terdapat *file* `A.class` dalam *folder* tersebut. Sehingga, *compiler* mencari *file* `A.java` dan mengkompilasinya.

Setelah *compiler* selesai mengkompilasi, kita dapat menjalankan program dengan menjalankan *file* `DemoPersegiPanjang.class` dengan perintah berikut.

```
java ClassTest
```

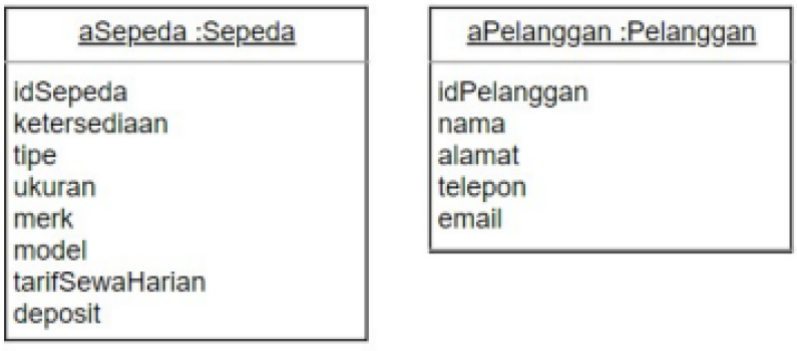
Output Program (ClassTest.java)

```
10  20
5   7
```

3.1.2 *Object*

Object merupakan konsep yang paling penting dalam pengembangan aplikasi yang berorientasikan *object*. Setiap *object* memiliki *state (attribute)* dan *behavior* masing-masing. Sebagai contoh, jika kita melihat *object* kucing, maka *state (attribute)* dari *object* tersebut adalah nama, jenis, dan warna dari kucing tersebut. Sedangkan *behavior* (perilaku) dari *object* tersebut adalah mengeong, bergoyang-goyang, dan berjalan. Sebuah *object* pada aplikasi akan menyimpan data-data dan *behavior* (perilaku) ditunjukkan melalui sebuah *method*. Jadi dalam pengembangan aplikasi, *object* dapat berkomunikasi dengan *object* yang lain melalui sebuah *method*. Seluruh *object* yang dimiliki oleh suatu *class* akan memiliki *attribute* yang sama, *behavior* (perilaku) yang sama, dan hubungan yang sama pula.

Sebagai contoh kasus, akan dibahas kasus sistem penyewaan sepada dalam penerapannya pada aplikasi. Pada kasus ini, sepeda dan pelanggan adalah *object* dari sistem penyewaan sepeda. Sedangkan *state (attribute)* dari sepeda adalah `idSepeda`, `ketersediaan`, `tipe`, `ukuran`, `merk`, `model`, `tarifSewaHarian`, dan `deposit`. Lalu *state (attribute)* dari pelanggan adalah `idPelanggan`, `nama`, `alamat`, `telepon`, dan `email`. Berikut diagram UML yang merepresentasikan sebuah *object*.



Pada diagram UML di atas, bagian atas adalah nama dari *object* yaitu `aSepeda :Sepeda` dan `aPelanggan :Pelanggan`. Sedangkan bagian bawah adalah nilai *attribute* dari *object* tersebut. Nama *object* selalu ditulis dengan garis bawah dan dapat memiliki dua bagian. Bagian pertama dari nama *object*, `aSepeda` dan `aPelanggan`, berfungsi untuk melabeli *object*; bagian kedua, `:Sepeda` dan `:Pelanggan`, mengidentifikasikan sebagai *object* dari *class* `Sepeda` dan `Pelanggan`. Namun pada diagram UML, *object* bisa digambarkan dengan *attribute* atau tanpa *attribute*.

Selain kegunaannya untuk menyimpan data, *object* merupakan sebuah konsep, abstraksi, atau hal yang memiliki batasan yang jelas, dan memiliki arti dalam area aplikasi yang dikerjakan. Kita menggunakan *object* untuk memodelkan karakteristik dunia nyata dari suatu area aplikasi dan untuk memberikan kita dasar dari implementasi komputer. Setiap *object* pada sistem memiliki tiga karakteristik: *behavior* (perilaku), *state* (keadaan), dan *identity* (identitas).

1. ***Behavior***

   Konsep *object* mirip dengan entitas pada *entity-relationship modelling*. Akan tetapi, tidak seperti entitas, *object* tidak hanya menyimpan informasi, *object* juga memiliki *behavior* (perilaku). Secara historis, alasan mengapa *object* memiliki *behavior* adalah karena *object* pada awalnya digunakan dalam simulasi komputer. Sebagai contoh, *object* `:Sepeda` mengetahui (menyimpan) tipenya, tarif sewa harian, dan deposit. Ketika kita mendesain *object* `:Sepeda`, kita menentukan hal apa yang *object* dapat lakukan. Sistem harus mampu menyimpan, meng-*update*, dan menampilkan nilai dari setiap *attribute* pada *object* `:Sepeda`. Pada pengembangan aplikasi berbasis *object*, proses yang bekerja pada *item* data terkait akan digabungkan bersama data tersebut. Proses ini disebut sebagai operasi. Perilaku dari suatu *object* dibagi menjadi banyak operasi, yang setiap operasinya merepresentasikan sesuatu yang *object* dapat lakukan. Contohnya, meng-*update* `tarifSewaHarian` dan menampilkan `deposit`. Perilaku dari suatu *object* dipicu sebagai tanggapan atas pesan yang dikirim dari *object* lain yang memintanya untuk melakukan salah satu operasinya.

2. ***State***

   Kebanyakan *object* memiliki *attribute*, seperti `tipe`, `tarifSewaHarian`, dan `deposit`. *Attribute* ini memiliki nilai, seperti `tipe` = sepeda gunung, `tarifSewaHarian` = Rp 50.000/hari, dan `deposit` = Rp 30.000. Umumnya nilai *attribute* dapat berubah, `tarifSewaHarian` bisa naik atau kita dapat mengubah jumlah `deposit`. *State* (keadaan) suatu *object* ditentukan berdasarkan nilai dari *attribute*-nya dan hubungannya dengan *object-object* lainnya. Alasan mengapa kita berfokus kepada keadaan *object* adalah karena perilaku dari *object* tersebut bisa bervariasi tergantung dari keadaan *object* tersebut. Contohnya, pada *object* `:Sepeda`, terdapat beberapa tipe sepeda seperti sepeda gunung, ataupun sepeda listrik. Tentu besaran nilai *attribute* dari masing-masing tipe akan berbeda satu sama lainnya yang mana akan menyebabkan *state* (keadaan) yang berbeda pula untuk masing-masing tipe sepeda.

3. ***Identity***

   Arti dari *object* memiliki *identity* (identitas) adalah setiap *object* memiliki sifat yang unik. Setiap *object* memiliki eksistensi yang terpisah dalam penyimpanan komputer. Setiap sepeda dalam sistem penyewaan sepeda direpresentasikan oleh *object* yang berbeda di dalam kodenya. Bahkan dua *object* yang memiliki nilai yang mirip dianggap berbeda dengan satu sama lain. *Object* harus dipanggil berdasarkan nama *object* tersebut. Contohnya, pada sistem penyewaan sepeda terdapat 600 sepeda yang disimpan pada array `jumlahSepeda[600]`. Apabila kita ingin mengetahui tarif sewa harian salah satu sepeda yang ada pada *array* tersebut, kita harus mengirimkan pesan yang ditujukan kepada sepeda tersebut (dalam contoh ini, sepeda nomor 105 pada *array*) dan memintanya untuk menampilkan tarif sewa harian dari sepeda nomor 105. Kita bisa menggunakan *keyword* `jumlahSepeda[105].tampilkanTarifSewaHarian()`. Nama dari *object* `:Sepeda` yang kita inginkan adalah `jumlahSepeda[105]`, kemudian `tampilkanTarifSewaHarian()` adalah operasi yang ingin kita jalankan.

3.1.3 *Field Instance* dan *Method Instance*

Object yang terbentuk dari *class* umumnya disebut sebagai ***instance*** (wujud) dari *class* tersebut. Dalam satu *class*, kita membuat lebih dari satu *instance*. Masing-masing dari *instance* tersebut akan memiliki *state*. *State* adalah nilai-nilai *field* dari suatu *instance* dalam satu waktu.

Sebagai contoh, berikut adalah *class* `Segitiga`.

Definisi Class (Segitiga.java)

```
/*
     Class Segitiga
*/
public class Segitiga
{
    private double alas;
    private double tinggi;

    /*
        Method setAlas menyimpan sebuah nilai
        dalam field alas.
        @param als Nilai yang disimpan dalam field alas.
    */
    public void setAlas(double als)
    {
        alas = als;
    }

    /*
        Method setTinggi menyimpan sebuah nilai
        dalam field tinggi.
        @param tng Nilai yang disimpan dalam field tinggi.
    */
    public void setTinggi(double tng)
    {
        tinggi = tng;
    }

    /*
        Method getAlas mengembalikan alas dari
        object Segitiga.
        @return Nilai dalam field alas.
    */
    public double getAlas()
    {
        return alas;
    }

    /*
        Method getTinggi mengembalikan tinggi dari
        object Segitiga.
        @return Nilai dalam field tinggi.
    */
    public double getTinggi()
    {
        return tinggi;
    }
```

```
    /*
        Method getLuas mengembalikan luas dari
        object Segitiga.
        @return Hasil dari setengah kali alas kali tinggi.
    */
    public double getLuas()
    {
        return 0.5 * alas * tinggi;
    }

}
```

Kemudian, program berikut membuat tiga *instance* dari *class* `Segitiga` yang masing-masing *instance* mempunyai *state-state* tersendiri:

Program (LuasSegitiga.java)

```
import java.util.Scanner;

/*
    Program ini membuat tiga instance dari
    class Segitiga.
*/
public class LuasSegitiga
{
    public static void main(String[] args)
    {
        double totalLuas;
        Scanner keyboard = new Scanner(System.in);

        // Buat tiga object Segitiga
        Segitiga luas1 = new Segitiga();
        Segitiga luas2 = new Segitiga();
        Segitiga luas3 = new Segitiga();

        // Dapatkan dimensi dari Segitiga pertama
        System.out.print("Masukkan alas Segitiga pertama: ");
        luas1.setAlas(keyboard.nextDouble());

        System.out.print("Masukkan tinggi Segitiga pertama: ");
        luas1.setTinggi(keyboard.nextDouble());

        // Dapatkan dimensi Segitiga kedua
        System.out.print("Masukkan alas Segitiga kedua: ");
        luas2.setAlas(keyboard.nextDouble());

        System.out.print("Masukkan tinggi Segitiga ketiga: ");
        luas2.setTinggi(keyboard.nextDouble());

        // Dapatkan dimensi Segitiga ketiga
        System.out.print("Masukkan alas Segitiga ketiga: ");
        luas3.setAlas(keyboard.nextDouble());

        System.out.print("Masukkan tinggi Segitiga ketiga: ");
        luas3.setTinggi(keyboard.nextDouble());
```

```
        // Tampilkan total luas
        totalLuas = luas1.getLuas() + luas2.getLuas()
                    + luas3.getLuas();

        System.out.println("Total luas dari tiga segitiga adalah " + totalLuas);
    }
}
```

Output Program (LuasRuangan.java)

```
Masukkan alas Segitiga pertama: 14
Masukkan tinggi Segitiga pertama: 12
Masukkan alas Segitiga kedua: 24
Masukkan tinggi Segitiga ketiga: 17
Masukkan alas Segitiga ketiga: 15
Masukkan tinggi Segitiga ketiga: 28
Total luas dari tiga segitiga adalah 498.0
```

Pada baris 15, 16, dan 17 di kode program `LuasSegitiga.java` kita mempunyai *statement-statement* berikut:

```
Segitiga luas1 = new Segitiga();
Segitiga luas2 = new Segitiga();
Segitiga luas3 = new Segitiga();
```

Tiga *statement* di atas membuat tiga *object* yang dimana setiap *object* adalah sebuah *instance* dari *class* `Segitiga`. Setelah program menginstansiasi tiga *instance* dari *class* `Segitiga`, program meminta pengguna untuk memasukkan nilai-nilai ke *field* `alas` dan field `tinggi` dari masing-masing *instance*. Pada contoh *output*, kita memasukkan nilai 14 dan 12 sebagai alas dan tinggi dari segitiga pertama, nilai 24 dan 17 sebagai alas dan tinggi dari segitiga kedua, dan nilai 15 dan 28 sebagai alas dan tinggi dari segitiga ketiga.

Masing-masing *instance* dari *class* `Segitiga` mempunyai variabel `panjang` dan `lebar` tersendiri. Variabel-variabel ini disebut sebagai **variabel *instance*** atau ***field instance***. Setiap *instance* dari sebuah *class* mempunyai *field-field instance* masing-masing dan dapat menyimpan nilai-nilainya sendiri dalam *field-field* tersebut.

Method-method yang beroperasi pada sebuah *instance* dari *class* disebut sebagai ***method instance***. Semua *method-method* dalam *class* `Segitiga` adalah *method instance* karena mereka melakukan operasi-operasi pada *instance* tertentu dari *class*. Sebagai contoh, lihat *statement* berikut yang ada pada baris 25 dari program `LuasSegitiga.java`.

```
luas1.setAlas(angka);
```

Statement di atas memanggil *method* `setAlas` pada *object* `luas1`. *Statement* ini menyebabkan *field* `alas` dalam *object* `luas1` ditugaskan dengan nilai `angka`. Sekarang perhatikan *statement* berikut pada baris 35 dari program `LuasSegitiga.java`:

```
luas2.setAlas(angka);
```

Statement di atas memanggil *method* `setAlas` pada *object* `luas2`. *Statement* ini menyebabkan *field* `alas` dari *object* `luas2` ditugaskan dengan nilai `angka`. Hal yang sama juga dilakukan oleh *statement* pada baris 45 dari program `LuasSegitiga.java`:

```
luas3.setAlas(angka);
```

Statement di atas memanggil *method* `setAlas` pada *object* `luas3`. *Statement* ini menyebabkan *field* `alas` dari *object* `luas3` ditugaskan dengan nilai `angka`.

3.1.4 *Constructor*

Constructor (pengkonstruksi) adalah *method* yang secara otomatis dipanggil ketika sebuah *object* dibuat. *Constructor* umumnya melakukan proses inisialisasi seperti menginisialisasi *field-field instance* ke suatu nilai.

Pada *class* `A` sebelumnya, diperlukan dua *statement* untuk membuat sebuah *object* `A` dan menginisialisasi nilai-nilai `field` dengan memanggil *method* `setData` , seperti contoh berikut:

```
A a1 = new A();
a1.setData(10, 20);
```

Akan lebih efisien jika dapat menginisialisasi *object* `A` pada saat pembuatannya. Sehingga dapat menyingkat proses pembuatan *object* `A` dan penginisialisasian nilai field `x` dan `y` dalam satu *statement*, seperti contoh berikut:

```
A a1 = new A(10, 20);
```

Kita dapat membuat *class* `A` melakukan inisialisasi saat proses pembuatan *object-object*nya dengan mendefinisikan sebuah *method constructor* yang menerima argumen. Kita mendefinisikan *method constructor* dengan menuliskan definisi *method* yang mempunyai nama yang sama dengan nama *class*. Sebagai contoh, kode berikut menambahkan sebuah *method constructor* pada *class* `A`.

Definisi Class (A.java)

```
/*
    Class A Versi 2 (dengan constructor).
*/
public class A
{
    private int x;
    private int y;

    /*
        Constructor
        @param x1 Nilai dari x.
        @param y1 Nilai dari y.
    */
    public A(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

// ...Kode-kode lain sama seperti sebelumnya.
```

Baris 14 sampai dengan 18 pada kode *class* `A` di atas adalah *constructor* dari *class* `A`:

```
public A(int x1, int y1)
{
        x = x1;
        y = y1;
}
```

Constructor ini menerima dua argumen, yang diberikan melalui variabel parameter `x1` dan `y1`. Nilai kedua variabel parameter ini kemudian ditugaskan ke *field* `x` dan field `y`.

Perhatikan pada *header* dari *constructor*, kita tidak menuliskan tipe *return* apapun (tidak juga `void`). Ini karena *constructor* tidak diperuntukkan untuk dieksekusi melalui pemanggilan *method* dan tidak dapat mengembalikan nilai. Berikut adalah contoh *statement* yang mendeklarasikan variabel `a1`, membuat sebuah object `A` dengan *field* `x` ditetapkan dengan nilai 10 dan *field* `y` ditetapkan dengan nilai 20:

```
A a1 = new A(10, 20);
```

Program berikut mendemonstrasikan pembuatan *object* `A` dengan *constructor*:

Program (DemoConstructor.java)

```
/*
    Program ini mendemonstrasikan constructor
    dari class PersegiPanjang.
*/
public class DemoConstructor
{
    public static void main(String[] args)
    {
        // Buat sebuah object A, berikan 20
        // dan 40 sebagai argumen ke constructor
        A a1 = new A(20, 40);

        // Buat sebuah object A, berikan 25
        // dan 75 sebagai argumen ke constructor
        A a2 = new A(25, 75);

        // Tampilkan nilai x1 dan y1
        System.out.println("Nilai x1 dan y1 adalah:");
        a1.display();

        // Tampilkan nilai x2 dan y2
        System.out.println("Nilai x2 dan y2 adalah:");
        a2.display();
    }
}
```

Output Program (DemoConstructor.java)

```
Nilai x1 dan y1 adalah:
20  40
Nilai x2 dan y2 adalah:
25  75
```

Tabel berikut menunjukkan perbedaan antara *constructor* dengan *method*.

	Constructor	**Method**
Tujuan	Digunakan untuk membuat dan menginisialisasi sebuah *object*	Digunakan untuk mengeksekusi beberapa *statement* tertentu
Return Type	*Constructor* tidak boleh menerima nilai *return*	*Method* dapat menerima nilai *return*
Object	*Constructor* menginisialisasi *object* yang tidak ada	*Method* hanya dapat dipanggil pada *object* yang ada
Penamaan	*Constructor* harus memiliki nama yang sama dengan *class* dimana *constructor* tersebut dibuat	Penamaan *method* dapat berupa apapun, tidak harus sesuai dengan nama *class*
Inheritance	*Constructor* tidak dapat diwariskan ke *subclass*	*Method* dapat diwariskan ke *subclass*

Pada pemrograman Java, *constructor* dibagi menjadi 3 tipe:

1. ### *Constructor Default*

   Apabila sebuah *class* tidak memiliki *constructor*, maka *compiler* Java akan otomatis membuat *constructor* pada *class* tersebut ketika dikompilasi. *Constructor* ini disebut dengan ***constructor default.*** *Constructor* ini tidak menerima argumen dan akan menginisialisasi *field* pada *object* dengan nilai 0 untuk tipe-tipe numerik (seperti tipe `int` dan `double`) dan nilai `null` untuk tipe-tipe data berupa *object* (seperti tipe `String`).

2. ### *Constructor* tanpa Argumen

   Terdapat sebuah *constructor* yang tidak menerima argumen apapun. Berikut adalah contoh menuliskan *constructor* tanpa argumen ke *class* `A`:

   ```
   public A()
   {
       x = 2;
       y = 3;
   }
   ```

   Berikut adalah kode dari definisi *class* `A` dengan sebuah *constructor* tanpa argumen:

   Definisi Class (A.java)

   ```
   /*
        Class A dengan constructor tanpa argumen
   */
   public class A
   {
       private int x;
       private int y;

       // Constructor tanpa argumen.
       // Menginisialisasi field x dan y
       // ke 2 dan 3.
       public A()
   ```

```
    {
      x = 2;
      y = 3;
    }

    public void setData(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    public void display()
    {
        System.out.println(x + "\t" + y);
    }
}
```

Kode berikut adalah contoh untuk membuat sebuah *object* dari *class* `A` dengan sebuah constrcutor tanpa argumen:

```
// Sekarang class A mempunyai constructor tanpa argumen
A a1 = new A();    // Memanggil constructor tanpa argumen
```

Setelah *statement* di atas dieksekusi, *object* dari *class* `A` yang direferensikan oleh variabel `a1`, nilai-nilai dari *field* `x` dan `y` dari *object* tersebut akan ditetapkan ke 2 dan 3.

3. ### *Constructor* dengan Argumen

Constructor juga dapat menerima satu atau lebih argumen. Sebagai contoh, kita dapat menuliskan *constructor* berikut ke *class* `A`:

```
public A(int x1, int y1)
{
    x = x1;
    y = y1;
}
```

Berikut adalah kode dari definisi *class* `A` dengan sebuah *constructor* dengan argumen.

Definisi Class (A.java)

```
/*
     Class A dengan constructor dengan argumen
*/
public class A
{
    private int x;
    private int y;

    // Constructor dengan argumen.
    // Menginisialisasi field x dan y
    // dengan nilai argumen yang diberikan.
    public A(int x1, int y1)
    {
        x = x1;
        y = y1;
```

```
    }

    public void setData(int x1, int y1)
    {
        x = x1;
        y = y1;
    }

    public void display()
    {
        System.out.println(x + "\t" + y);
    }
}
```

Untuk membuat sebuah *object* `A` dengan *constructor* di atas kita dapat menuliskan *statement* seperti berikut:

```
// Membuat sebuah object A dengan constructor dengan argumen
A a1 = new A(3, 4);
```

Setelah *statement* di atas dieksekusi, variabel `a1` akan mereferensikan sebuah *object* dari *class* `A` dengan *field* `x` menyimpan nilai `3` dan field `y` menyimpan nilai `4`.

3.1.5 Menggunakan *Object* dengan *Method*

1. Memberikan *Object* ke *Method*

Program berikut mendemonstrasikan sebuah *method* yang menerima argumen berupa referensi ke *object* `A`:

Program (PassObject.java)

```
/*
    Program ini memberikan sebuah object sebagai argumen
*/
public class PassObject
{
    public static void main(String[] args)
    {
        // Buat object A
        A a1 = new A(4, 7);

        // Berikan sebuah referensi ke object A
        // sebagai argumen ke method tampilkanNilaiXdanY.
        tampilkanNilaiXdanY(a1);
    }

    /*
        Method tampilkanNilaiXdanY menampilkan
        nilai x dan y.
        @param r Sebuah referensi ke sebuah object A.
    */
    public static void tampilkanNilaiXdanY(A nilai)
    {
        // Tampilkan panjang dan lebar.
        System.out.println("Nilai x dan y = ");
```

```
        nilai.display();
    }
}
```

Output Program (PassObject.java)

```
Nilai x dan y =
4   7
```

Dalam *method* `main` dari program di atas, variabel `a1` adalah sebuah variabel referensi ke *object* `A`. Pada baris 13, variabel `a1` diberikan sebagai argumen ke *method* `tampilkanNilaiXdanY`. Saat *method* ini dieksekusi, alamat ke *object* `A` yang diberikan melalui variabel `a1` disalin ke variabel parameter `nilai` yang juga merupakan variabel referensi ke *class* `A`. Ini berarti saat *method* `tampilkanNilaiXdanY` dieksekusi, variabel `a1` dan `nilai` keduanya mereferensikan *object* yang sama.

Selain itu *method* yang menerima referensi *object* sebagai argumen juga dapat memodifikasi isi dari *object* yang diterimanya. Berikut contoh kode *method* yang memodifikasi isi *object* yang diterimanya.

Program (PassObject2.java)

```
/*
    Program ini memberikan sebuah object sebagai argumen.
    Object tersebut dimodifikasi oleh method yang menerimanya.
*/
public class PassObject2
{
    public static void main(String[] args)
    {
        // Buat object A
        A a1 = new A(12, 5);

        // Tampilkan isi object
        System.out.println("Nilai X dan Y:");
        a1.display();

        // Berikan referensi ke object ke method ubahNilaiXdanY.
        ubahNilaiXdanY(a1);

        // Tampilkan isi object
        System.out.println("\nNilai X dan Y sekarang:");
        a1.display();

    }

    /*
        Method ubahNilaiXdanY menetapkan field x dan y
        dari object A ke 0, 0.
    */
    public static void ubahNilaiXdanY(A nilai)
    {
        nilai.setData(0, 0);
    }
}
```

Output Program (PassObject2.java)

```
Nilai X dan Y:
12  5

Nilai X dan Y sekarang:
0   0
```

2. Mengembalikan *Object* dari *Method*

Method juga dapat mengembalikan sebuah referensi ke suatu *object*. Sebagai contoh, program berikut menggunakan *class* `A` dan menggunakan *method* `buatNilai` yang mengembalikan sebuah referensi ke *object* `A`:

Program (ReturnObject.java)

```
import java.util.Scanner;
/*
    Program ini mendemonstrasikan bagaimana sebuah method
    dapat mengembalikan sebuah referensi ke object.
*/
public class ReturnObject
{
    public static void main(String[] args)
    {
        A nilai;

        // Buat object A melalui method buatNilai
        nilai = buatNilai();

        // Tampilkan object A nilai
        nilai.display();
    }

    /*
        Method buatNilai membuat sebuah object A
        dengan nilai x dan y yang diberikan pengguna.
        @return Sebuah referensi ke object.
    */
    public static A buatNilai()
    {
        int x;   // Untuk menyimpan x.
        int y;  // Untuk menyimpan y.

        // Dapatkan x dan y dari pengguna
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Masukkan x: ");
        x = keyboard.nextInt();
        System.out.print("Masukkan y: ");
        y = keyboard.nextInt();

        // Buat sebuah object A dan
        // kembalikan referensi ke object tersebut.
        return new A(x, y);
    }
}
```

Output Program (ReturnObject.java)

```
Masukkan x: 4
Masukkan y: 6
4   6
```

Perhatikan *method* `buatNilai` mempunyai tipe *return* `A`. *Method* yang mempunyai tipe *return* `A` berarti *method* tersebut mengembalikan sebuah referensi ke *object* `A`. *Statement* berikut, pada baris 14 dari program di atas, menugaskan nilai *return* dari *method* `buatNilai` ke variabel `nilai`:

```
nilai = buatNilai();
```

Setelah *statement* di atas dieksekusi, variabel `p` akan mereferensikan *object* `A` yang dikembalikan oleh *method* `buatNilai`.

Sekarang mari kita lihat *method* `buatNilai`. *Method* ini meminta pengguna memasukkan `x` dan `y` dari `A`, lalu menggunakan nilai `x` dan `y` yang diberikan untuk membuat sebuah *object* `A`. *Statement* terakhir dari *method* `buatNilai`:

```
return new A(x, y);
```

Staement ini menggunakan *keyword* `new` untuk membuat sebuah *object* `A`, memberikan `x` dan `y` sebagai argumen ke *constructor*. Alamat dari *object* yang dibuat ini kemudian dikembalikan dari *method* `buatNilai`.

3.1.6 *Array* dari *Object*

Misalkan terdapat sebuah *class* bernama `Mahasiswa` yang didefinisikan seperti pada kode berikut:

Definisi Class (Mahasiswa.java)

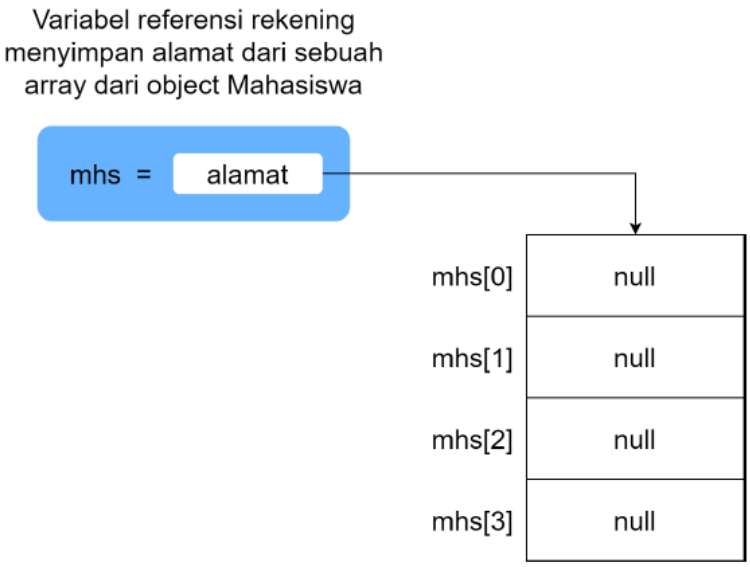
```
/*
    Class Mahasiswa mendaftarkan nama mahasiswa dan jurusan
    mahasiswa.
*/
public class Mahasiswa
{
    public String nama;     // Nama mahasiswa
    public String jurusan;  // Jurusan mahasiswa

    /*
        Constructor ini menetapkan nama dan jurusan dengan field
        kosong
    */
    public Mahasiswa()
    {
        nama = "";
        jurusan = "";
    }

    /*
        Constructor ini menetapkan nama dan jurusan
```

```
        ke nilai yang diberikan sebagai argumen.
        @param n Nama mahasiswa.
        @param jrsn Jurusan mahasiswa.
    */
    public Mahasiswa(String n, String jrsn)
    {
        nama = n;
        jurusan = jrsn;
    }

    /*
        Method display menampilkan nama dan jurusan mahasiswa
        yang sudah terdaftar
    */
    public void display()
    {
        System.out.println("Nama mahasiswa: " + nama);
        System.out.println("Jurusan: " + jurusan);
    }
}
```

Dari kode di atas, kita dapat membuat *array* yang berisi *object-object* dari *class* `Mahasiswa`. Berikut adalah caranya.

```
final int BANYAK_MAHASISWA = 2;
Mahasiswa[] mhs = new Mahasiswa[BANYAK_MAHASISWA];
```

Variabel yang mereferensikan *array* dari *object* `Mahasiswa` bernama `mhs`. Sama seperti *array* dari `String`, setiap elemen dari *array* ini adalah sebuah variabel referensi. Berikut merupakan gambar ilustrasinya.



Setiap elemen dari *array* diinisialisasi ke nilai `null`. Nilai `null` menandakan bahwa elemen-elemen *array* belum mereferensikan *object*. Kode berikut menggunakan *loop* untuk membuat *object-object* `Mahasiswa` untuk setiap elemen:

```
for (int index = 0; index < mhs.length; index++)
{
    mhs[index] = new Mahasiswa();
}
```

Pada kode di atas, pembuatan *object* `Mahasiswa` dilakukan melalui *constructor* tanpa argumen. Ingat, *constructor* tanpa argumen dari *class* `Mahasiswa` yang kita tulis sebelumnya menugaskan variabel `nama` dan `jurusan` memiliki nilai kosong. Setelah *loop* dieksekusi, setiap elemen dari *array* `mhs` akan mereferensikan sebuah *object* `Mahasiswa`. *Object-object* dalam sebuah *array* diakses menggunakan notasi *subscript*. Program berikut mendemonstrasikan penggunaan *array* dari *object-object*.

Program (ArrayObject.java)

```
import java.util.Scanner;

/*
    Program ini bekerja dengan sebuah array dari
    tiga object Mahasiswa.
*/
public class ArrayObject
{
    public static void main(String args[])
    {
        final int BANYAK_MAHASISWA = 2;
        String nama;
        String jurusan;

        Mahasiswa[] mhs = new Mahasiswa[BANYAK_MAHASISWA];

        Scanner keyboard = new Scanner(System.in);

        for (int index = 0; index < mhs.length; index++)
        {
            // Dapatkan nama dan jurusan mahasiswa
            System.out.print("Masukkan nama mahasiswa Ke-" +
                             (index + 1) + ": ");
            nama = keyboard.nextLine();
            System.out.print("Masukkan jurusan mahasiswa Ke-" +
                             (index + 1) + ": ");
            jurusan = keyboard.nextLine();

            System.out.println("");
            mhs[index] = new Mahasiswa(nama, jurusan);
        }

        System.out.println("Daftar Mahasiswa");

        for (int index = 0; index < mhs.length; index++)
        {
            mhs[index].display();
        }
    }
}
```

Output Program (ArrayObject.java)

```
Masukkan nama mahasiswa Ke-1: Riki Mahendra
Masukkan jurusan mahasiswa Ke-1: Informatika

Masukkan nama mahasiswa Ke-2: Dini Mariana
Masukkan jurusan mahasiswa Ke-2: Teknik Mesin

Daftar Mahasiswa
Nama mahasiswa: Riki Mahendra
Jurusan: Informatika
Nama mahasiswa: Dini Mariana
Jurusan: Teknik Mesin
```

3.2 *Static Class Member*

Setiap *instance* dari *class* mempunyai variabel-variabel tersendiri yang disebut sebagai *instance variables*. Kita dapat membuat sejumlah *instance* dari suatu *class* dan setiap *instance* akan mempunyai *instance variables* tersendiri. Sebagai contoh, misalkan variabel `nilai` mereferensikan sebuah *instance* dari *class* `A` dan kita mengeksekusi *statement* berikut:

```
nilai.setData(10, 20);
```

Statement di atas menyimpan nilai 10 ke *variable* `x` dan nilai 20 ke *variable* `y` dari *instance* yang direferensikan oleh `nilai`.

Sebuah *class* juga dapat mempunyai *instance method*. *Instance method* adalah *method* yang bekerja pada suatu *instance* dari sebuah *class* dimana *method* tersebut didefinisikan. Ketika kita memanggil *instance method*, *method* tersebut melakukan suatu operasi pada *instance* tertentu dari *class* tersebut. Sebagai contoh, misalkan variabel `boks` mereferensikan sebuab *instance* dari *class* `A` dan kita mempunyai *statement* berikut:

```
x = nilai.getData();
```

Statement di atas memanggil *method* `getData`, yang mengembalikan nilai dari *variable* `x` dan `y` yang dimiliki oleh *instance* dari *class* `A` yang direferensikan oleh variabel `nilai`. *Instance variables* dan *instance method* keduanya terasosiasikan ke sebuah *instance* tertentu dari suatu *class*, dan mereka tidak dapat digunakan sampai sebuah *instance* dari *class* dibuat.

Kita dapat membuat sebuah *variable* atau *method* yang bukan merupakan bagian dari *instance* dari suatu *class*. Member-member yang bukan merupakan bagian dari *instance* disebut sebagai ***static member***. Misalkan, kita dapat membuat *static variable* dalam sebuah *class*. *Static variable* ini tidak disimpan dalam sebuah *instance* dari *class* tersebut. *Variable* ini tidak memerlukan sebuah *instance* untuk dapat menyimpan suatu nilai. Kita juga dapat membuat *static method* pada sebuah *class*. *Static method* tidak bekerja pada *variables* dari *instance* tertentu dari *class* yang memiliki *method* tersebut. Kita tidak perlu membuat sebuah *instance* dari *class* tersebut untuk memanggil *static method*.

3.2.1 *Static Variables*

Static variables dideklarasikan dengan menuliskan *keyword* `static`. *Static variable* dapat dibagikan ke seluruh *class* yang diinstansiasi. *Static variable* merupakan variabel yang tidak melekat pada *object* instansiasi akan tetapi melekat pada *class*. Ketika sebuah variabel dideklarasikan sebagai `static`, hanya ada satu salinan variabel tersebut dalam memori meskipun terdapat lebih dari satu *instance* dari *class* tempat variabel tersebut dideklarasikan. Sebagai contoh, *class* `MenghitungMahasiswa` berikut menggunakan variabel `static` untuk menghitung banyaknya *instance* yang dibuat.

Definisi Class (MenghitungMahasiswa.java)

```
/*
    Class ini mendemonstrasikan static variable.
*/
public class MenghitungMahasiswa
{
    private static int hitungMahasiswa = 0;

    /*
        Constructor ini menginkrementasi static variable
        hitungMahasiswa. Ini untuk menghitung banyaknya
        instance dari class ini yang dibuat.
    */
    public MenghitungMahasiswa()
    {
        hitungMahasiswa++;
    }

    /*
        Method getHitungMahasiswa mengembalikan
        banyaknya instance dari class ini yang telah dibuat.
        @return Nilai dalam variable hitungMahasiswa.
    */
    public int getHitungMahasiswa()
    {
        return hitungMahasiswa;
    }
}
```

Pada baris 6, kita mendeklarasikan sebuah *static variable* bernama `hitungInstance` dan menginisialisasinya ke 0:

```
private static int hitungMahasiswa = 0;
```

Kita mendeklarasikan *static variable* dengan menuliskan *keyword* `static` setelah *access specifier*, yaitu `private`, dan sebelum tipe data dari *variable*, yaitu `int`.

Pada baris 13 sampai dengan 16, kita menuliskan *constructor* dari *class* `MenghitungMahasiswa`. *Constructor* ini menggunakan operator inkrementasi `++` untuk menginkrementasi variabel `hitungMahasiswa`. Ini berarti, setiap kali sebuah *instance* dari *class* `Menghitung` dibuat, *constructor* ini akan dipanggil dan variabel `hitungMahasiswa` akan diinkrementasi. Sehingga, variabel `hitungInstance` akan berisi banyaknya *instance* dari *class* `MenghitungMahasiswa` yang telah dibuat.

Pada baris 23 sampai dengan 26 adalah definisi *instance method* bernama `getHitungMahasiswa`. *Method* ini mengembalikan nilai yang disimpan dalam `hitungMahasiswa`. Program berikut mendemonstrasikan *class* `Menghitung`.

Program (DemoStatic.java)

```
/*
    Program ini mendemonstrasikan class MenghitungMahasiswa.
*/
public class DemoStatic
{
    public static void main(String[] args)
    {
        int banyakMahasiswa;

        // Buat tiga instance dari class MenghitungMahasiswa
        MenghitungMahasiswa object1 = new MenghitungMahasiswa();
        MenghitungMahasiswa object2 = new MenghitungMahasiswa();
        MenghitungMahasiswa object3 = new MenghitungMahasiswa();

        // Dapatkan banyaknya instance melalui
        // static variable dari class
        banyakMahasiswa = object1.getHitungMahasiswa();
        System.out.println("Terdapat " + banyakMahasiswa +
                           " mahasiswa di kelas 1IA10 yang"+
                           " sudah mengisi absen.");
    }
}
```

Output Program (DemoStatic.java)

```
Terdapat 3 mahasiswa di kelas 1IA10 yang sudah mengisi absen.
```

Program di atas membuat tiga *instance* dari *class* `MenghitungMahasiswa` dan mereferensikannya dengan variabel `object1`, `object2`, dan `object3`. Meskipun terdapat tiga *instance* dari *class*, hanya terdapat satu *static variable*.

Pada baris 17, program di atas memanggil *method* `getHitungMahasiswa` untuk mendapatkan banyaknya *instance* yang telah dibuat:

```
banyakMahasiswa = object1.getHitungMahasiswa();
```

Meskipun program ini memanggil *method* `getHitungMahasiswa` dari `object1`, nilai yang sama akan dikembalikan jika program ini memanggilnya dari *object-object* lainnya.

3.2.2 *Static Method*

Static method merupakan *method* yang eksekusinya tidak melekat pada *object* instansiasi akan tetapi melekat pada *class*. Dideklarasikan dengan *keyword* `static`. Perhatikan definisi *class* berikut:

Definisi Class (Metrik.java)

```
/*
    Class ini mendemonstrasikan static methods.
```

```
*/
public class Metrik
{
    /*
        Method meterKeInch mengkonversi jarak dalam meter
        ke inch.
        @param  Jarak dalam meter.
        @return Jarak dalam inch.
    */
    public static double meterKeInch(double m)
    {
        return m * 39.37;
    }

    /*
        Method inchKeMeter mengkonversi jarak dalam inch
        ke meter.
        @param km Jarak dalam inch.
        @return Jarak dalam meter.
    */
    public static double inchKeMeter(double inch)
    {
        return inch / 39.37;
    }
}
```

Static method didefinisikan dengan menuliskan *keyword* `static` setelah *access specifier* pada *header method*. *Class* `Metrik` di atas mempunyai dua *static method*, yaitu `meterKeInch` dan `inchKeMeter`. Karena, kedua *method* ini *static*, keduanya dapat dipanggil tanpa harus membuat suatu *instance* dari *class* `Metrik` terlebih dahulu. Kita dapat memanggil *static method* dengan menuliskan nama *class* yang diikuti dengan titik sebelum nama *method* dan argumen-argumen dalam tanda kurung. Berikut adalah contoh pemanggilan method `meterKeInch`:

```
m = Metrik.meterKeInch(20.0);
```

`Metrik.meterKeInch(20.0)` adalah pemanggilan *method* `meterKeInch` dalam *class* `Metrik` dengan memberikan nilai 10 sebagai argumen ke *method* tersebut. Nilai *return* dari pemanggilan *method* ini lalu ditugaskan ke variabel `m`. Perhatikan pada *statement* di atas, *static method* `meterKeInch` tidak dipanggil dari sebuah *instance* dari *class* `Metrik`, tetapi dipanggil langsung dari *class* `Metrik`. Program berikut mendemonstrasikan penggunaan *class* `Metrik`:

Program (DemoMetrik.java)

```
import java.util.Scanner;

/*
    Program ini mendemonstrasikan penggunaan
    static methods dari class Metric.
*/
public class DemoMetrik
{
    public static void main(String[] args)
    {
        double inch;
        double meter;
```

```
        // Buat object Scanner untuk mendapatkan input keyboard
        Scanner keyboard = new Scanner(System.in);

        // Dapatkan jarak dalam mil.
        System.out.print("Masukkan jarak dalam meter: ");
        meter = keyboard.nextDouble();

        // Konversi jarak dari mil ke km dan tampilkan hasilnya.
        inch = Metrik.meterKeInch(meter);
        System.out.printf("%.2f meter = %.2f inch.\n", meter, inch);

        // Dapatkan jarak dalam inch.
        System.out.print("Masukkan jarak dalam inch: ");
        inch = keyboard.nextDouble();

        // Konversi jarak dari km ke mil dan tampilkan hasilnya.
        meter = Metrik.inchKeMeter(inch);
        System.out.printf("%.2f inch = %.2f m.\n", inch, meter);
    }
}
```

Output Program (DemoMetrik.java)

```
Masukkan jarak dalam meter: 10.0
10.00 meter = 393.70 inch.
Masukkan jarak dalam inch: 24.0
24.00 inch = 0.61 m.
```

Static method umumnya digunakan untuk membuat *utility classes* yang melakukan operasi-operasi kalkulasi terhadap data tetapi tidak memerlukan untuk menyimpan data tersebut. *Class* `Math` yang sudah tersedia dalam Java adalah contoh *utility class*. *Class* `Math` tidak perlu diinstansiasi terlebih dahulu untuk menjalankan *method* di dalamnya.

3.3 *Class Relationship*

Hubungan antar *class* dengan *class* yang lain dalam sebuah sistem, secara umum dapat dikategorikan menjadi empat, yaitu:

3.3.1 *Association*

Asosiasi adalah hubungan yang bisa saling menggunakan di dalam sebuah *class*, dan tidak saling memiliki. Asosiasi diantara *class-class* artinya ada hubungan antara *object-object* pada *class-class* yang berhubungan. Banyaknya *object* yang terhubung tergantung dengan beragamnya *object* (*multiplicity*) yang ada asosiasi.

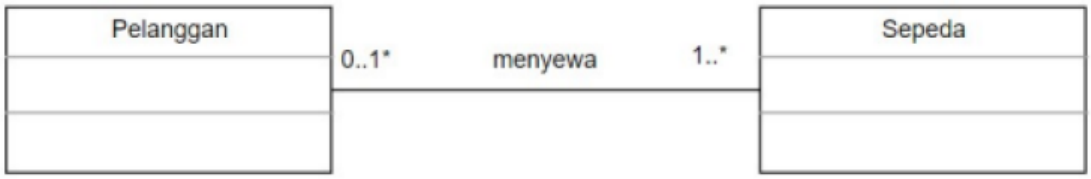
1. ***Multiplicity***

   Pada relasi terdapat suatu penanda yang disebut *multiplicity*. *Multiplicity* ditandai dengan angka dan asteris pada garis. *Multiplicity* ini akan mengindikasikan berapa banyak *object* dari suatu *class* terelasi ke *object* lain.

   Notasi UML untuk *multiplicity* ini adalah sebagai berikut:

Arti	Contoh	Notasi
Angka yang persis	Persis satu	1 (dapat tidak ditulis)
	Persis enam	6
Banyak	Nol atau lebih Satu atau lebih, banyak	0..* 1..*, *
	Satu atau lebih, banyak	1..*, *
Jangkauan spesifik	Satu sampai empat, nol sampai 6	1..4, 0..6
Pilihan	Dua atau empat atau lima	2, 4, 5

Sebagai contoh pada kasus sistem penyewaan sepeda, diagram di bawah ini menunjukkan relasi antara *class* `Pelanggan` dan `Sepeda`, yang mana diantaranya terdapat *class* `Sewa` untuk penghubung antara *class* `Pelanggan` dan `Sepeda`.



Kita bisa menginterpretasi relasi antara `Pelanggan` dan `Sepeda` pada diagram *class* di atas sebagai berikut: pertama-tama, kita bisa memperhatikan banyak `:Sepeda`, yang merupakan *instance* dari *class* `Sepeda`, yang bisa dipinjam oleh `:Pelanggan`, yang merupakan *instance* dari *class* `Pelanggan`, dan banyak `:Pelanggan` yang meminjam `:Sepeda`. Agar kita bisa mengetahui berapa banyak `:Sepeda` yang `:Pelanggan` bisa pinjam, kita bisa memperhatikan angka pada ujung asosiasi *class* `Sepeda`. Angka tersebut adalah 1..*. Ini berarti pelanggan dapat menyewa satu atau lebih sepeda. Dalam istilah *object*, diagram tersebut menjelaskan bahwa satu `:Pelanggan` dapat dihubungkan dengan lebih dari satu `:Sepeda`.

2. ## Asosiasi *Relationship*

Asosiasi *Relationship* yaitu menggambarkan suatu *class* yang mengirimkan pesan ke *class* lain, memungkinkan suatu *class* mengetahui *attribute* dan operasi yang mempunyai *access specifier public* dari *class* lain. Tipe - tipe hubungan dalam asosiasi *relationship* ada tiga yaitu *one to one relationship*, *one to many relationship*, dan *many to many relationship*.

1. ***One to One Relationship***

   One to one relationship (1:1) adalah apabila satu *occurance* (anggota) suatu entiti hanya berhubungan dengan tidak lebih dari satu *occurance* lawannya. Sebagai misal perhatikan contoh berikut.

   Seorang suami hanya dibolehkan mempunyai satu istri dan seorang istri juga hanya dibolehkan mempunyai seorang suami. Maka hubungan antara suami dan istri adalah hubungan satu-satu (1:1). Dari contoh ini ada dua entiti yang berbeda yaitu, suami dan istri, dan hubungan kedua entiti tersebut dapat dejelaskan atau dinamai istri dipunyai oleh, atau mempunyai seorang Suami. Hubungan satu-satu seperti yang diterangkan diatas akan terlihat seperti gambar berikut ini.

Pemberian nama hubungan dapat ditinjau dari dua titik. Seperti contoh diatas dapat dikatakan suami "mempunyai" istri atau istri "dipunyai" suami. Pemberian nama suatu hubungan ini tidaklah hal yang prinsipil, kedua alternatif diatas dapat diterima. Apa bila ada hanya satu hubungan antara dua buah entiti, hubungan dapat saja dinamai dengan gabungan nama dua buah entiti yang berhubungan.

Berikut ini adalah kode penerapan hubungan *one to one relationship* pada Java.

Definisi Class (Suami.java)

```
public class Suami
{
    // Attribute namaSuami
    private String namaSuami;

    // Constructor class Suami.
    public Suami(String nama)
    {
        namaSuami = nama;
    }

    public String getNamaSuami()
    {
        return namaSuami;
    }

    public void setNamaSuami(String nama)
    {
        namaSuami = nama;
    }

    // Method toString()
    @Override
    public String toString()
    {
        return "{" +" namaSuami='" + getNamaSuami() + "'" +"}";
    }
}
```

Definisi Class (Istri.java)

```
public class Istri
{
    // Attributes
    private String namaIstri;
    private String tglMenikah;

    // Attribute namaSuami merupakan relasi ke class Suami
    private Suami suami;

    // Constructor class Istri
    public Istri(String nama, String tgl, Suami aSuami)
    {
        namaIstri = nama;
        tglMenikah = tgl;
        suami = aSuami;
    }
```

```
    public String getNamaIstri()
    {
        return namaIstri;
    }

    public void setNamaIstri(String nama)
    {
        namaIstri = nama;
    }

    public String getTglMenikah()
    {
        return tglMenikah;
    }

    public void setTglMenikah(String tgl)
    {
        tglMenikah = tgl;
    }

    public Suami getSuami()
    {
        return suami;
    }

    public void setSuami(Suami aSuami)
    {
        suami = aSuami;
    }

    // Operasi toString()
    @Override
    public String toString()
    {
        return "{" +
            " namaIstri='" + getNamaIstri() + "'" +
            ", namaSuami='" + suami.getNamaSuami() + "'" +
            ", tglMenikah='" + getTglMenikah() + "'" +
            "}";
    }
}
```

Program (Main.java)

```
/*
    Program ini mendemonstrasikan hubungan
    one to one relationship. Berikut adalah
    main class-nya.
*/
public class Main
{
    public static void main(String[] args)
    {
        // Object suami
        Suami suami = new Suami("Catur");
        System.out.println(suami.toString());
```

```
        // Object istri yang berkaitan dengan object suami
        Istri istri = new Istri("Dama","1 April 2022",suami);
        System.out.println(istri.toString());
    }
}
```

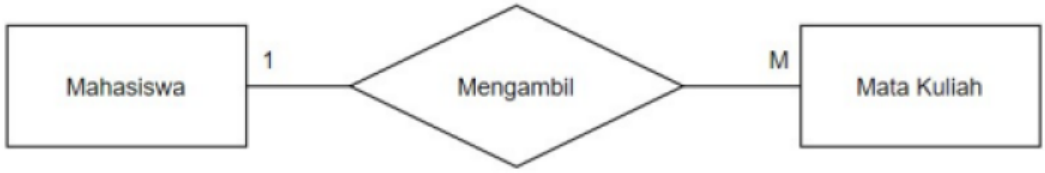
Output Program (Main.java)

```
{ namaSuami='Catur'}
{  namaIstri='Dama',  namaSuami='{  namaSuami='Catur'}',  tglMenikah='1
April 2022'}
```

2. ***One to Many Relationship***

   One to many relationship (1:M) apabila satu *occurance* dari suatu entiti hanya berhubungan dengan tidak lebih dari satu *occurance* entiti lawan, tetapi satu *occurance* entiti lawan dapat berhubungan dengan lebih dari satu *occurance* entiti tersebut. Perhatikanlah contoh berikut.

   Setiap mahasiswa di Universitas Gunadarma dapat mengambil banyak matkul dalam satu semester. Contoh tersebut dapat diidentifikasi dua entiti yaitu, mahasiswa dan mata kuliah, dan hubungan kedua entiti tersebut mengambil atau diambil. Secara grafik hubungan *One to Many* (1:M) ini dapat dilihat seperti dibawah ini.



Berikut ini adalah kode penerapan hubungan *one to many relationship* pada Java.

Definisi Class (Matkul.java)

```
public class Matkul
{
    // Attributes
    private String kdMatkul;
    private String nmMatkul;

    // Constructor class Matkul
    public Matkul(String aKdMatkul, String aNmMatkul)
    {
        kdMatkul = aKdMatkul;
        nmMatkul = aNmMatkul;
    }

    public String getKdMatkul()
    {
        return kdMatkul;
    }

    public void setKdMatkul(String aKdMatkul)
    {
        kdMatkul = aKdMatkul;
    }
```

```
    public String getNmMatkul()
    {
        return nmMatkul;
    }

    public void setNmMatkul(String aNmMatkul)
    {
        nmMatkul = aNmMatkul;
    }
}
```

Definisi Class (Mahasiswa.java)

```
public class Mahasiswa
{
    // Attributes
    private String NIM;
    private int jmlMatkul=0;
    private Matkul[] matkuls;

    // Constructor class Mahasiswa
    public Mahasiswa(String aNIM, int aJmlMatkul, Matkul[] aMatkuls) {
        NIM = aNIM;
        jmlMatkul = aJmlMatkul;
        matkuls = aMatkuls;
    }

    public String getNIM()
    {
        return NIM;
    }

    public void setNIM(String aNIM)
    {
        NIM = aNIM;
    }

    public int getJmlMatkul()
    {
        return jmlMatkul;
    }

    public void setJmlMatkul(int aJmlMatkul)
    {
        jmlMatkul = aJmlMatkul;
    }

    public Matkul[] getMatkuls()
    {
        return matkuls;
    }

    public void setMatkuls(Matkul[] aMatkuls)
    {
        matkuls = aMatkuls;
    }
}
```

Program (Mengambil.java)

```
/*
    Program ini mendemonstrasikan hubungan
    one to many relationship. Berikut adalah
    main class-nya.
*/
public class Mengambil
{
    public static void main(String[] args)
    {
        // Inisialisasi list dari object Matkul yang dinamakan matkuls
        Matkul[] matkuls = new Matkul[2];

        // Inisialisasi object matkul1 dan matkul2
        matkuls[0] = new Matkul("TIT306","Pemrograman Robotik");
        matkuls[1] = new Matkul("TIT304",
                                "Pemrograman Berorientasi Objek");

        // Object mahasiswa yang berkaitan dengan
        // object matkul1 dan matkul2
        Mahasiswa mahasiswa = new Mahasiswa("2009-51-100"
                                            ,matkuls.length,matkuls);

        // Tampilkan data mahasiswa
        System.out.println("NIM = " + mahasiswa.getNIM());
        System.out.println("Jumlah Mata Kuliah = "
                           + mahasiswa.getJmlMatkul());
        System.out.println("Mata Kuliah yang diambil: ");
        for (Matkul mk: mahasiswa.getMatkuls())
        {
            System.out.print("Kode MK = " + mk.getKdMatkul());
            System.out.print(", ");
            System.out.print("Nama MK = " + mk.getNmMatkul());
            System.out.println();
        }
    }
}
```

Output Program (Mengambil.java)

```
NIM = 2009-51-100
Jumlah Mata Kuliah = 2
Mata Kuliah yang diambil:
Kode MK = TIT306, Nama MK = Pemrograman Robotik
Kode MK = TIT304, Nama MK = Pemrograman Berorientasi Objek
```

3. ***Many to Many Relationship***

   Sekarang perhatikan contoh berikut. Sebuah perusahaan mempunyai informasi tentang persediaan yang ada di gudang. Masing-masing persediaan dapat dibeli dari banyak atau lebih dari satu pemasok, dan hampir setiap pemasok mampu menyediakan lebih dari satu macam barang.

Dari contoh di atas kita melihat ada dua entitas yaitu, barang dan pemasok, dan hubungannya dapat dinamai memasok atau dipasok oleh. Secara sederhana dapat dikatakan bahwa satu persediaan dipasok oleh banyak penjual dan satu penjual memasok banyak barang. Diagram untuk contoh diatas adalah sebagai berikut.

Berikut ini adalah kode penerapan hubungan *one to many relationship* pada Java.

Definisi Class (Pemasok.java)

```
public class Pemasok
{
    // Attributes
    private int idPemasok;
    private String namaPemasok;
    private Barang[] barangDipasok;

    // Constructor class Pemasok
    public Pemasok(int aIdPemasok, String aNamaPemasok)
    {
        idPemasok = aIdPemasok;
        namaPemasok = aNamaPemasok;
    }

    public int getIdPemasok()
    {
        return idPemasok;
    }

    public void setIdPemasok(int id)
    {
        idPemasok = id;
    }

    public String getNamaPemasok()
    {
        return namaPemasok;
    }

    public void setNamaPemasok(String nama)
    {
        namaPemasok = nama;
    }

    public Barang[] getBarangTersedia()
    {
        return barangDipasok;
    }

    public void setBarangTersedia(Barang[] daftarBarang) {
        barangDipasok = daftarBarang;
    }
}
```

Definisi Class (Barang.java)

```
public class Barang
{
    // Attributes
 private int idBarang;
    private String namaBarang;
    private Pemasok[] daftarPemasok;

    // Constructor Persediaan
    public Barang(int id, String nama)
    {
        idBarang = id;
        namaBarang = nama;
    }

    public int getIdBarang()
    {
        return idBarang;
    }

    public void setIdBarang(int id)
    {
        idBarang = id;
    }

    public String getNamaBarang()
    {
        return namaBarang;
    }

    public void setNamaBarang(String nama)
    {
        namaBarang = nama;
    }

    public Pemasok[] getPemasok()
    {
        return daftarPemasok;
    }

    public void setPemasok(Pemasok[] daftar)
    {
        daftarPemasok = daftar;
    }
}
```

Program (DataPersediaan.java)

```
public class DataPersediaan
{
    public static void main(String[] args)
 {
        // Buat 3 list pemasok
        Pemasok[] pemasokMinyak = new Pemasok[2];
        Pemasok[] pemasokBeras = new Pemasok[2];
        Pemasok[] pemasokGula = new Pemasok[2];
```

```
        // Buat 3 object Pemasok: pemasok1, pemasok2, dan pemasok3
        Pemasok pemasok1 = new Pemasok(001, "Jaya Abadi");
        Pemasok pemasok2 = new Pemasok(002, "Tinggi Rejeki");
        Pemasok pemasok3 = new Pemasok(003, "Rejeki Makmur");

        // Memasukkan pemasok1, pemasok2, dan pemasok3 ke dalam
        // pemasokMinyak, pemasokBeras, pemasokGula
        pemasokMinyak[0] = pemasok1;
        pemasokMinyak[1] = pemasok2;

        pemasokBeras[0] = pemasok1;
        pemasokBeras[1] = pemasok3;

        pemasokGula[0] = pemasok2;
        pemasokGula[1] = pemasok3;

        Barang[] daftarBarang = new Barang[3];

        daftarBarang[0] = new Barang(001, "Minyak");
        daftarBarang[1] = new Barang(002, "Beras");
        daftarBarang[2] = new Barang(003, "Gula");

        // set pemasok dari masing masing persediaan
        daftarBarang[0].setPemasok(pemasokMinyak);
        daftarBarang[1].setPemasok(pemasokBeras);
        daftarBarang[2].setPemasok(pemasokGula);

        // Tampilkan Persediaan
        System.out.println("============PERSEDIAAN============");
        for (Barang brg: daftarBarang)
        {
            System.out.print("ID = " + brg.getIdBarang());
            System.out.print(", ");
            System.out.println("Nama Barang = " + brg.getNamaBarang());
            System.out.println("Pemasok: ");
            for (Pemasok pmsk: brg.getPemasok())
            {
                System.out.print("ID Pemasok = " + pmsk.getIdPemasok());
                System.out.print(", ");
                System.out.println("Nama Pemasok = "
                                   + pmsk.getNamaPemasok());
            }
            System.out.println();
        }
    }
}
```

Output Program (DataPersediaan.java)

```
============PERSEDIAAN============
ID = 1, Nama Barang = Minyak
Pemasok:
ID Pemasok = 1, Nama Pemasok = Jaya Abadi
ID Pemasok = 2, Nama Pemasok = Tinggi Rejeki

ID = 2, Nama Barang = Beras
```

```
Pemasok:
ID Pemasok = 1, Nama Pemasok = Jaya Abadi
ID Pemasok = 3, Nama Pemasok = Rejeki Makmur

ID = 3, Nama Barang = Gula
Pemasok:
ID Pemasok = 2, Nama Pemasok = Tinggi Rejeki
ID Pemasok = 3, Nama Pemasok = Rejeki Makmur
```

Dapat dilihat pada hasil program, disana dapat diketahui siapa saja pemasok untuk masing-masing jenis barang. Begitu pula untuk pemasok, dapat diketahui setiap jenis barang yang tersedia dari masing-masing pemasok.

3.3.2 *Dependency*

Dependency merupakan hubungan antar *class*, dimana salah satu *class* nya bergantung (*dependent*) dengan *class* lain. Namun ketergantungan tersebut tidak timbal balik. Sebagai contoh, perhatikan class `Customer` dan class `Order` berikut:

Definisi Class (Customer.java)

```
public class Customer
{
    private String id;
    private String nama;

    public Customer(String aId, String aNama)
    {
        id = aId;
        nama = aNama;
    }

    public String getId()
    {
        return id;
    }

    public String getNama()
    {
        return nama;
    }
}
```

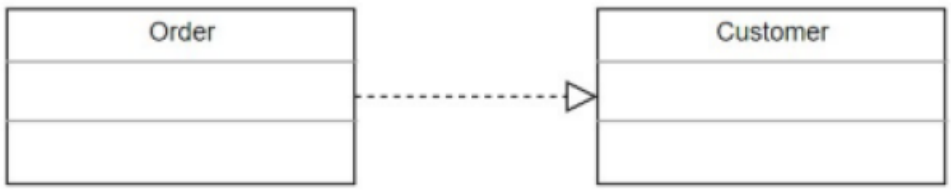
Definisi Class (Order.java)

```
public class Order
{
    private String orderId;
    private String customerId;
    private double total;

    public void setOrderId(String aOrderId)
    {
        orderId = aOrderId;
    }
```

```
    public String getOrderId()
    {
        return orderId;
    }

    public void setcustomerId(Customer customer)
    {
        customerId = customer.getId();
    }

    public String getCustomerId()
    {
        return customerId;
    }

    public void setTotal(double aTotal)
    {
        total = aTotal;
    }

    public double getTotal()
    {
        return total;
    }
}
```

Dapat dilihat pada dua kode di atas, class `Order` *dependent* (bergantung) kepada *class* `Customer`. Ketergantungan *class* `Order` kepada *class* `Customer` terlihat pada kode *method* `setCustomerId` dimana sebuah *class* `Customer` diperlukan untuk menetapkan nilai *field* `customerId`.

Relasi *dependency* ini digambarkan dengan panah yang dari satu *class* ke *class* lainnya. Arah panah menunjukkan *class* yang dibutuhkan. Diagram dari *class* `Customer` dan *class* `Order` dapat dilihat pada gambar di bawah ini.



Selain *dependency* terjadi di hubungan antar *class*, *dependency* juga dapat terjadi ketika sebuah *class* bergantung pada *class* itu sendiri. Hal ini dapat terjadi karena sebuah *object* dalam *class* tersebut berinteraksi dengan *object* lain yang berada dalam *class* yang sama. Contoh yang paling sering dijumpai adalah ketika melakukan concat `String` seperti berikut:

```
String str1, str2, str3;
str1 = “Hello”;
str2 = “World”;
str3 = str1.concat(str2)
```

Object `str1` bergantung dengan `str2` ketika melakukan *concatenation*.

3.3.3 *Aggregation*

Aggregation merupakan hubungan antar *class*, dimana sebuah *class* disusun oleh *class* yang lain. Relasi agregasi dapat diidentifikasi dengan ciri-ciri berikut:

- Jika frasa seperti 'terdiri atas', 'memiliki sebuah', atau 'bagian dari' digunakan untuk menjelaskan relasi tersebut atau sering disebut dengan "*HAS-A Relationship*".
- *Aggregation* adalah *unidirectional* asosiasi atau **relasi satu arah**. Contoh, *department* dapat memiliki siswa tetapi tidak sebaliknya.
- Pada *Aggregation*, kedua *entries* **dapat berdiri sendiri**, yang artinya mengakhiri satu entitas tidak akan memengaruhi entitas lainnya.

Gambar di bawah ini menggambarkan relasi *Aggregation* antara mahasiswa dan alamat.

Dalam notasi UML, hubungan agregasi digambarkan dengan *diamond* putih yang ditempelkan pada *class* yang memiliki, dan tidak dibubuhkan panah pada ujung yang tidak memiliki simbol *diamond* putih. *Multiplicity* dapat dituliskan pada ujung relasi, sama seperti notasi pada asosiasi.

Dilihat pada gambar *class* diagram di atas, bahwa *class* `Mahasiswa` memiliki *class* `Alamat` sebagai variabel nya. Meskipun *class* `Mahasiswa` memiliki *class* `Alamat`, namun kedua *class* tersebut dapat dibuat secara independen. Berikut penulisan kode untuk *class* `Mahasiswa` dan `Alamat`.

Definisi Class (Alamat.java)

```
public class Alamat
{
    // attributes
    private String namaJalan;
    private String namaKota;

    // constructor
    public Alamat(String jalan, String kota)
    {
        namaJalan = jalan;
        namaKota = kota;
    }

    // mengembalikan deskripsi objek alamat
    public String toString()
    {
        String hasil;
        hasil = namaJalan + "\n" + namaKota;
        return hasil;
    }
}
```

Definisi Class (Mahasiswa.java)

```
public class Mahasiswa
{
```

```
    // attributes
    private String namaDepan;
    private String namaBelakang;
    private Alamat alamatRumah;
    private Alamat alamatKampus;

    // constructor
    public Mahasiswa(String depan, String belakang, Alamat rumah, Alamat kampus)
    {
        namaDepan = depan;
        namaBelakang = belakang;
        alamatRumah = rumah;
        alamatKampus = kampus;
    }

    // mengembalikan deskripsi objek mahasiswa
    public String toString ()
    {
        String hasil;
        hasil = namaDepan + " " +namaBelakang + "\n" +
                "Alamat rumah: \n" + alamatRumah + "\n" +
                "Alamat kampus: \n" +alamatKampus;
        return hasil;
    }
}
```

Program (TestMahasiswa.java)

```
public class TestMahasiswa
{
    public static void main(String[] args)
    {
        // membuat objek alamat untuk memberi nilai namaJalan dan namaKota
        Alamat kampus = new Alamat("Jl. Margonda Raya","Depok");
        Alamat rumah = new Alamat("Jl. Asem No. 70","Depok");

        /* membuat objek mahasiswa untuk memberi nilai namaDepan
           dan namaBelakang serta set alamat rumah dan kampus
           dengan objek rumah dan kampus
        */
        Mahasiswa mhs1 = new Mahasiswa("Lisa", "Marlina", rumah, kampus);
        System.out.println(mhs1);
    }
}
```

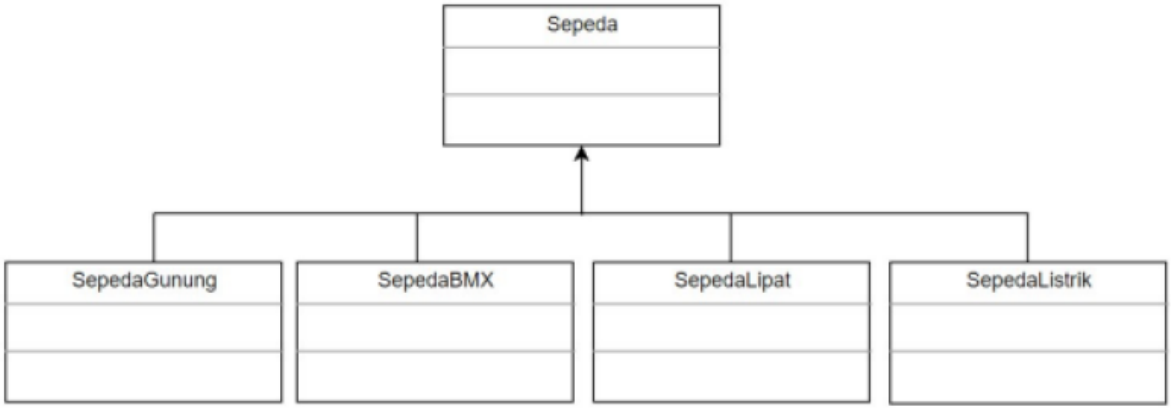
Output Program (TestMahasiswa.java)

```
Lisa Marlina
Alamat rumah:
Jl. Asem No. 70
Depok
Alamat kampus:
Jl. Margonda Raya
Depok
```

Bisa dilihat bahwa pada *class* Jurusan, menggunakan *class* Mahasiswa sebagai variabel di dalamnya, sehingga bisa dibilang bahwa *class* Jurusan memiliki *class* Mahasiswa. Apabila *class* Mahasiswa dihapus maka *class* Jurusan tidak dapat berdiri, namun tidak terjadi sebaliknya, *class* Mahasiswa masih bisa berdiri tanpa terdapat *class* Jurusan. Hubungan seperti ini disebut *aggregation*.

3.3.4 *Inheritance*

Pada materi sebelumnya, ketika kita membuat beberapa *class*, *class-class* tersebut dapat memiliki beberapa *attribute* dan operasi yang sama. Kita dapat menambahkan *class* baru yang memiliki fitur-fitur yang sama tersebut dan meninggalkan hanya fitur-fitur yang berbeda kepada *class* aslinya. Proses ini disebut sebagai generalisasi. Perhatikan diagram berikut!



Pada contoh diagram di atas, *class* `SepedaGunung`, `SepedaBMX`, `SepedaLipat`, dan `SepedaListrik` adalah spesialisasi dari *class* general, yaitu *class* `Sepeda`. Relasi antara *class* general dan spesialisasinya disebut sebagai relasi *inheritance* (pewarisan). Mekanisme *inheritance* adalah membuat *class* spesialisasi dapat membagikan atau mewariskan fitur-fitur dari *class* general. Notasi UML untuk relasi *inheritance* adalah panah yang menunjuk dari *class* spesialisasi ke *class* general.

Relasi *inheritance* dapat dijelaskan sebagai berikut:

- *Class* spesialisasi mewariskan dari *class* general
- *Subclass* mewarisi dari *superclass*
- *Class* anak mewarisi dari *class* orangtua
- *Class* turunan mewarisi dari *class* asal

Ketika kita membuat *class* spesialisasi, *class* tersebut mewarisi semua *attribute*, operasi, dan relasi dari *parent class*-nya. *Inheritance* dapat dengan mudah digunakan dengan keliru. Suatu kumpulan *class* seharusnya tidak dihubungkan, kecuali terdapat relasi *is-a* atau *is-a-kind-of* di antaranya.

Suatu relasi generalisasi atau *inheritance* dapat diidentifikasi dengan ciri-ciri berikut:

- Jika frasa seperti '*is-a*' atau '*is-a-kind-of*' dapat digunakan untuk mendeksripsikan relasi di antara *class* tersebut, contohnya kuda memiliki relasi '*is-a*' dengan mamalia.
- Di mana satu atau lebih *class* memiliki *attribute* dan operasi yang serupa.

Generalisasi dan *inheritance* merupakan teknik yang berguna karena kita dapat menggunakan ulang *class* yang sudah ada. Mengklasifikasikan *class* menjadi hierarki *inheritance* berarti kita dapat menghindari menulis ulang kode. Operasi yang diwariskan berada pada *superclass* dan *subclass* tidak perlu memiliki operasi versinya sendiri, kecuali *subclass* akan digunakan untuk menspesialisasikan operasi tersebut. Ini berarti jika kita ingin mengubah kode yang mengimplementasikan operasi, kita hanya perlu mengubahnya sekali di *superclass*. Penjelasan lebih lanjut mengenai *inheritance* akan dibahas pada Bab selanjutnya.

3.4 *Method Design*

Setelah mempelajari bagaimana mengidentifikasi *class* dan *object*, desain dari setiap *method* akan mempengaruhi bagaimana *class* tersebut didefinisikan dan kebiasaannya.

3.4.1 *Method Decomposition*

Terkadang setiap *object* memiliki fungsi yang tidak dapat dijadikan dalam sebuah *method* saja. Sehingga, kita perlu menguraikan sebuah *method* tersebut menjadi beberapa *method* agar membuat *object* yang lebih mudah dipahami. Dekomposisi *method* merupakan proses pemecahan *method* besar menjadi beberapa bagian. Dekomposisi *method* dilakukan untuk meningkatkan *reusability* pada struktur kode program. Dekomposisi *method* juga dilakukan untuk membuat *method* menjadi lebih ringkas dan mudah dibaca.

```
public double hitungUpah(int jmlHariKerja, boolean isMarried)
{
    // attributes
    double upahHarian = 100000;
    double uangMakanHarian = 50000;
    double tunjanganMenikah = 500000;
    double totalUpahKotor;
    double totalUpahBersih;
    double tunjanganTotal;

    tunjanganTotal = jmlHariKerja * uangMakanHarian;
    if (isMarried)
        tunjanganTotal += tunjanganMenikah;

    totalUpahKotor = (upahHarian * jmlHariKerja) + tunjanganTotal;
    totalUpahBersih = totalUpahKotor - (totalUpahKotor * 0.05);

    return totalUpahBersih;
}
```

Dalam kode sumber di atas, terdapat pendeklarasian variabel `totalUpahBersih` dan `tunjanganTotal`. Jika diperhatikan kembali dalam program, variabel tersebut digunakan secara berulang. Jika program diatas dilakukan dekomposisi *method*, maka hasil kode program akan menjadi:

```
public double hitungUpah(int jmlHariKerja, boolean isMarried)
{
    // attributes
    double upahHarian = 100000;
    double uangMakanHarian = 50000;
    double tunjanganMenikah = 500000;
    double totalUpahKotor;

    totalUpahKotor = hitungUpahPokok(jmlHariKerja, upahHarian) +
                     hitungTunjangan(jmlHariKerja, isMarried,
                     uangMakanHarian, tunjanganMenikah);

    return hitungUpahBersih(totalUpahKotor);

}
```

Hasil dekomposisi *method* dari program di atas adalah *method* `hitungUpahPokok`, `hitungTunjangan`, dan `hitungUpahBersih`. Dari dekomposisi *method* pada program di atas, menghasilkan kode sumber yang lebih ringkas dan efisien. *Method* yang didekomposisi dipecah menjadi beberapa bagian seperti berikut:

```
private double hitungUpahBersih(double totalUpahKotor)
{
    return totalUpahKotor - (totalUpahKotor * 0.05);
}

private double hitungTunjangan(int jmlHariKerja, boolean isMarried,
       double uangMakanHarian, double tunjanganMenikah)
{
    double tunjanganTotal = jmlHariKerja * uangMakanHarian;
    if (isMarried)
        tunjanganTotal += tunjanganMenikah;
    return tunjanganTotal;
}

private double hitungUpahPokok(int jmlHariKerja, double upahHarian)
{
    return upahHarian * jmlHariKerja;
}
```

Setiap *method* menyimpan variabel dan fungsi nya masing-masing. Sehingga ketika masing-masing fungsi akan dieksekusi, hanya perlu dipanggil masing-masing *method* dan tidak perlu lagi memanggil variabel nya secara berulang-ulang.

3.4.2 *Method Parameters*

Dilihat dari sudut pandang prosesnya parameter pada *method* di Java dapat dikategorikan menjadi 2 yaitu *actual parameter* dan *formal parameter*.

1. *Actual parameter* : merupakan parameter yang dikirim saat *method* dipanggil. Dilihat dari sudut pandang program yang memanggil *method* tersebut.

   ```
   int luasPersegi = hitungLuasPersegi(panjang * lebar);
   ```

   Operasi variabel panjang dikalikan dengan variabel `lebar` merupakan *actual parameter* yang diberikan untuk method `hitungLuasPersegi`.

2. *Formal parameter* : merupakan parameter yang di definisikan pada *header method*. Dilihat dari sudut pandang *method* yang akan di eksekusi.

   ```
   public int hitungLuasPersegi(int panjang, int lebar){
       return panjang * lebar;
   }
   ```

   Variabel `int panjang` dan `int lebar` yang diberikan ketika membuat *method* `hitungLuasPersegi`, merupakan *formal parameter*.

Seluruh eksekusi *method* pada Java mengirimkan parameter dari *actual parameter* ke *formal parameter* berdasarkan nilai (*passed by-value*). Sehingga ketika terjadi perubahan nilai parameter melalui *formal parameter* pada ekseskusi *method* perubahan tersebut tidak akan berpengaruh pada nilai *actual parameter*.

Pada kode program di bawah ini, variabel `panjang` dan `lebar` memiliki nilai masing-masing yaitu `10` dan `5`. Nilai ini tidak akan berubah atau berpengaruh ketika dipanggil variabel nya ke dalam *actual parameter*.

```
int panjang = 10;
int lebar = 5;
int luasPersegi = hitungLuasPersegi(panjang * lebar);
```

```
public int hitungLuasPersegi(int panjang, int lebar){
    panjang = 20;
    lebar = 10;
    return panjang * lebar;
}
```

Namun, ketika `panjang` dan `lebar` diberikan nilai dalam *formal parameter*, maka nilai dari masing-masing variabel akan berubah. Sehingga pada kode program di atas, nilai `panjang` dan `lebar` akan berubah menjadi `20` dan `10`. Untuk parameter berupa *object* maka nilai yang dikirim berdasarkan rererensi alamat memorinya.

3.4.3 *Method Overloading*

Method overloading adalah bagian penting dalam *object oriented programming*. *Method overloading* adalah istilah ketika kita mempunyai lebih dari satu *method* dengan nama yang sama, namun menggunakan parameter-parameter dengan tipe-tipe data berbeda. Kita menggunakan *method overloading* ketika kita memerlukan sejumlah cara untuk melakukan operasi yang sama. Sebagai contoh, misalkan sebuah *class* mempunyai dua *method* berikut:

```
public int tambah(int a, int b)
{
    int jumlah = a + b;
    return tambah;
}

public String tambah(String str1, String str2)
{
    String kombinasi = str1 + str2;
    return kombinasi;
}
```

Kedua *method* di atas mempunyai nama `tambah`. Keduanya menerima dua argumen, yang keduanya dijumlahkan. *Method* pertama menerima dua argumen `int` dan mengembalikan jumlah kedua argumen tersebut. *Method* kedua menerima dua referensi `String` dan mengembalikan sebuah referensi ke sebuah `String` yang merupakan hasil konkatenasi dari dua argumen. Ketika kita memanggil *method* `tambah`, *compiler* harus menentukan *method* mana yang cocok dengan pemanggilan tersebut. Jika kita memanggil *method* `tambah` dengan argumen dua `int` maka *method* pertama yang didefinisikan dengan dua parameter `int` yang dieksekusi. Sedangkan jika kita memanggil *method* `tambah` dengan dua argumen `String`, *method* kedua yang didefinisikan dengan dua parameter `String` yang dieksekusi.

Bagaimana *compiler* Java menentukan *method* mana yang dieksekusi ketika *method* yang dioverloading dipanggil? *Compiler* Java menggunakan *signature* dari *method* untuk membedakan antara *method* bernama sama. *Signature* dari *method* terdiri dari nama *method* dan tipe data dari parameter-parameter *method*.

```
tambah(int, int)
tambah(String, String)
```

Perhatikan bahwa tipe *return method* bukan merupakan bagian *signature* dari *method*. Oleh karena ini, *method* `add` berikut tidak dapat ditambahkan pada *class* yang sama dengan *class* yang mempunyai dua *method* sebelumnya:

```
public int tambah(String str1, String str2)
{
    int jumlah = Integer.parseInt(str1) + Integer.parseInt(str2)
    return jumlah;
}
```

Perhatikan *method* di atas mempunyai *signature method* yang sama dengan *method* kedua di atas yang mempunyai dua parameter `String`. Karena *compiler* Java hanya melihat signature dari *method* untuk membedakan *method* bernama sama satu sama lainnya, maka *compiler* Java tidak bisa membedakan *method* ini dengan *method* kedua pada contoh di atas, sehingga sebuah *error* akan didapatkan ketika kita mencoba mengkompilasi *class* dengan ketiga *method* di atas.

3.5 Enkapsulasi

Enkapsulasi adalah sebuah peroses pemaketan / penyatu data bersama metode – metodenya, dimana hal ini bermanfaat untuk menyembunyikan rincian – rincian implementasi dari pemakai. Maksud dari enkapsulasi ini adalah untuk menjaga suatu proses program agar tidak dapat diakses secara sembarangan atau di intervensi oleh program lain. Konsep enkapsulasi sangat penting dilakukan untuk menjaga kebutuhan program agar dapat diakses sewaktu-waktu, sekaligus menjaga program tersebut.

Secara teknis enkapsulasi menyembunyikan data atau variabel dari *class* lain dan hanya dapat diakses oleh anggota fungsi dari *class* itu sendiri yang telah dideklarasikan. Pada enkapsulasi, data dalam *class* tersembunyi dari *class* lainnya, yang juga dikenal dengan istilah ***data-hiding***.

3.5.1 *Data Hiding* (Penyembunyian Data)

Data hiding (penyembunyian data) adalah konsep penting dalam pemrograman berorientasi *object*. *Object* harus didesain untuk menyembunyikan data internalnya dari kode di luar *class* yang membentuk *object* tersebut. Hanya *method-method* yang dituliskan dalam *class* yang mempunyai akses langsung dan dapat mengubah data internal dari *object*. Kita menyembunyikan data internal *object* dengan membuat *field-field class* menjadi `private` dan *membuat method-method* yang mengakses *field-field object* sebagai `public`.

Penggunaan *data hiding* sangat bermanfaat jika kita terlibat dalam sebuah tim yang mengembangkan *software* besar dan *programmer-programmer* lain menggunakan *class-class* yang kita tulis, dengan menyembunyikan data, kita dapat memastikan bahwa *class* yang kita desain digunakan dan bekerja sesuai dengan peruntukkan yang kita inginkan.

3.5.2 Accessor dan Mutator

Seperti yang telah disebutkan sebelumnya, *class* harus didesain dengan memperhatikan *data hiding*. *Data hiding* diterapkan dengan membuat *field-field class* `private` dan menyediakan *method-method* `public` untuk mengakses dan mengubah nilai-nilai pada *field-field* tersebut. *Data hiding* memastikan bahwa data dalam *object* konsisten dengan tujuan dari *object* tersebut.

Dalam pemrograman berorientasi *object* terdapat istilah-istilah yang digunakan untuk *method-method* yang mengakses *field* `private` dari *object*, yaitu ***accessor*** dan ***mutator***. *Method* yang mengembalikan nilai dalam sebuah *field* dan tidak mengubah nilai dari *field* disebut sebagai method accessor. Sedangkan *method* yang mengubah nilai *field* disebut sebagai method mutator. Pada class `PersegiPanjang`, method `setPanjang` dan `setLebar` adalah *method* mutator.

Umumnya *method-method* accessor dinamakan dengan awalan `get` yang diikuti dengan nama *field* dan *method-method* mutator dinamakan dengan awalan `set` yang diikuti dengan nama *field*. Sehingga *method* mutator terkadang disebut juga sebagai "*setter*" dan *method* accessor disebut juga sebagai "*getter*". Berikut merupakan contoh kode untuk enkapsulasi:

Definisi Class (Encapsulate.java)

```
// Program java untuk mendemonstrasikan enkapsulasi.

public class Encapsulate
{
    /* Deklarasi variabel private.
    hanya dapat diakses oleh public method dari class.
    */
    private String idName;
    private int idRoll;
    private int idAge;

    public int getAge()
    {
        return idAge;
    }

    public String getName()
    {
        return idName;
    }

    public int getRoll()
    {
        return idRoll;
    }

    public void setAge( int newAge)
    {
        idAge = newAge;
    }

    public void setName(String newName)
    {
        idName = newName;
    }

    public void setRoll( int newRoll)
    {
        idRoll = newRoll;
    }
}
```

Pada program di atas, *class* `Encaplulate` dienkapsulasi sebagai variabel yang dideklarasikan *private*. *Method* `getAge()`, `getName()`, dan `getRoll()` adalah *method* accesor yang diatur sebagai tipe *public*, *method* ini digunakan untuk mengakses variabel yang telah dienkapsulasi. Sedangkan *method setter* seperti `setName()`, `setAge()`, `setRoll()` adalah *method* mutator juga dideklarasikan sebagai *public* dan digunakan untuk mengatur nilai dari variabel.

Program yang digunakan untuk mengakses variabel dari *class* `Encapsulate` adalah sebagai berikut:

Program (TestEncapsulation.java)

```
public class TestEncapsulation
{
    public static void main (String[] args)
    {
        Encapsulate obj = new Encapsulate();

        /* atur nilai dari variabel */
        obj.setName("Sulaiman");
        obj.setAge(19);
        obj.setRoll(51);

        /* menampilkan nilai dari variabel */
        System.out.println("ID Nama: "+ obj.getName());
        System.out.println("ID Usia: "+ obj.getAge());
        System.out.println("ID Roll: "+ obj.getRoll());

        /* Mengakses nilai secara langsung ke variabel adalah
        tidak mungkin dalam enkapsulasi.
        */
    }
}
```

Output Program (TestEncapsulation.java)

```
ID Nama: Sulaiman
ID Usia: 19
ID Roll: 51
```

3.5.3 Keunggulan Enkapsulasi

Dengan melakukan enkapsulasi berupa *data hiding* dan penggunaan *accessor* dan *mutator*, *value* di setiap *attribute* dapat dikontrol secara penuh dan lingkungan luar tidak perlu tahu akan hal tersebut. Sebagai contoh *attribute age* dapat dikontrol agar *value*-nya tidak kurang dari 0 dan proses ini tidak perlu diketahui oleh lingkungan di luar kelas yang melakukan enkapsulasi. Selain itu, terdapat beberapa keunggulan enkapsulasi, diantaranya :

1. **Data tersembunyi (*data hiding*):** pengguna tidak mengetahui implementasi dasar dari *class*. Tidak mungkin bagi *user* untuk melihat nilai dari data yang disimpan pada variabel dalam *class*. Hal yang diketahui oleh *user* adalah nilai telah diatur melalui *setter method* dan variabel telah diinisialisasi dengan nilai tertentu.
2. **Meningkatkan fleksibilitas:** *programmer* dapat membuat variabel dari *class* sebagai *read-only* atau *write-only* tergantung pada kebutuhan. Jika variabel dibuat dalam bentuk *read-only* maka *method setter* seperti `setName()`, `setAge()`, dan lain sebagainya harus dihilangkan.

Pada contoh program sebelumnya jika ingin membuat program yang hanya bersifat *write-only* maka *method* `getName()`, `getAge()`, dan lain sebagainya harus dihilangkan.

3. ***Reusability*:** *Reusability* adalah kemampuan dapat digunakan kembali komponen perangkat lunak untuk mengurangi waktu, biaya, dan sumber daya manusia. *Reusability* yang dilakukan secara manual membutuhkan tenaga dan waktu lebih banyak, terlebih jika perangkat lunak yang diukur cukup kompleks. Pengukuran manual juga memungkinkan terjadinya kesalahan perhitungan metrik. Oleh karena itu, dibutuhkan sistem sebagai alat pembantu dari perhitungan manual. Dengan melakukan enkapsulasi, dapat meningkatkan *reusability* dan kemudahan dalam melakukan perubahan dengan *new requirement*.
4. **Kemudahan pengujian kode program:** enkapsulasi kode memudahkan proses pengetesan program untuk unit-unit programnya masing-masing.

3.6 *Interface*

3.6.1 *Abstract Class*

Abstract Class adalah kelas yang mengandung satu *method* abstrak atau lebih, *abstract class* tersebut digunakan hanya untuk membuat sebuah *method* yang tanpa ada implementasinya secara langsung.

Sebagai contoh, kita akan membuat 2 buah *class*, misalnya kita namakan *class* `Komponen` dan *class* `MembuatKue`, *class* `Komponen` akan kita ubah menjadi *abstract class* dan *class* `MembuatKue` adalah *class* utama yang mempunyai *method* main, nantinya *method* pada kelas komponen (abstrak) akan di turunkan pada *class* `MembuatKue`.

Definisi Class (Komponen.java)

```
// program abstract class
public abstract class Komponen
{
    // abstract method
    abstract void bahan_bahan();
    abstract void peralatan();
    abstract void proses_pembuatan();
}
```

Jadi intinya *abstract method* itu adalah sebuah *method* yang tidak tahu mau kita apakan nantinya, sebuah *class* dan *abstract method* dibuat oleh seorang programmer sebagai acuan atau gambaran dari program yang ingin mereka buat.

Definisi Class (MembuatKue.java)

```
public class MembuatKue extends Komponen
{
    public static void main(String[] args)
    {
    }

    @Override
    void bahan_bahan()
    {
    }
```

```
    @Override
    void peralatan()
    {
    }

    @Override
    void proses_pembuatan()
    {
    }
}
```

Class `MembuatKue.java` melakukan *extends class* `Komponen.java` dan mengimplementasikan semua *method* yang terdapat pada *class* `Komponen.java`. Dengan adanya *class* dan *abstract method*, programmer jadi tahu *statement* apa saja yang harus dibuatnya, kita dapat mengisi *statement* pada *method* tersebut seperti ini:

Definisi Class (MembuatKue.java)

```
public class MembuatKue extends Komponen
{
    public static void main(String[] args)
    {
        // Membuat Instance atau Object dari Class MembuatKue
        MembuatKue kue = new MembuatKue();
        kue.bahan_bahan();
        kue.peralatan();
        kue.proses_pembuatan();
    }

    @Override
    void bahan_bahan()
    {
        String bahan1 = "Tepung Terigu";
        String bahan2 = "Gula";
        String bahan3 = "Telur";
        System.out.println("====== BAHAN-BAHAN ======");
        System.out.println("1."+bahan1);
        System.out.println("2."+bahan2);
        System.out.println("3."+bahan3);
    }

    @Override
    void peralatan()
    {
        String alat1 = "Oven";
        String alat2 = "Mixer";
        String alat3 = "Loyang";
        System.out.println("====== ALAT-ALAT ======");
        System.out.println("1."+alat1);
        System.out.println("2."+alat2);
        System.out.println("3."+alat3);
    }

    @Override
    void proses_pembuatan()
    {
        System.out.println("====== PROSES ======");
```

```
        System.out.println("1.Aduk dan Campurkan Semua Bahan Pada Mixer");
        System.out.println("2.Masukan Pada Loyang");
        System.out.println("3.Oven Sampai Matang");
        System.out.println("4.Selesai");
    }
}
```

Output Program (MembuatKue.java)

```
====== BAHAN-BAHAN ======
1.Tepung Terigu
2.Gula
3.Telur
====== ALAT-ALAT ======
1.Oven
2.Mixer
3.Loyang
====== PROSES ======
1.Aduk dan Campurkan Semua Bahan Pada Mixer
2.Masukan Pada Loyang
3.Oven Sampai Matang
4.Selesai
```

3.6.2 *Interface*

Interface adalah *class* yang tidak memiliki tubuh pada *method-method*nya. *Method* pada *interface* tersebut harus diimplementasikan dalam kelas turunannya tidak boleh tidak. Di dalam *interface*, deklarasi variabel memiliki atribut *final* sehingga bersifat absolut. Keyword *final* inilah yang menjadi keunikan sendiri bagi *interface* bahwa *ouput* dari bagian *interface* berupa *final* yang tidak diganti pada saat implementasi kecuali di *override*.

Keuntungan membuat *interface* sendiri adalah menutupi kekurangan pada Java yang hanya memperbolehkan satu *class* saja yang berhak mendapatkan warisan kelas induk (*extends*). Sehingga satu *class* hanya dapat menggunakan satu *class* induk, sebaliknya pada *interface* dapat di implementasi lebih dari satu. Ciri-ciri *interface* adalah *interface* tidak dapat di instansiasi, tidak terdapat konstruktor dan semua *method* pada *interface* adalah abstrak. Tujuan *interface* adalah menerapkan teknik *polymorhpisme* pada Java. Jadi kelas turunan dapat bebas mengubah karakteristik yang ada.

Interface memiliki pengertian dan fungsi yang hampir sama dengan *abstract class*, walalupun fungsi dari keduanya sama, akan tetapi ada beberapa perbedaan. Berikut ini adalah tabel yang menjelaskan perbedaan diantara abstract dan interface:

Abstract	***Interface***
Bisa berisi *abstract* dan *non-abstract method*.	Hanya boleh berisi *abstract method*.
Kita harus menuliskan sendiri *modifier*nya.	Kita tidak perlu menulis *public abstract* di depan nama *method*. Karena secara implisit, *modifier* untuk *method* di *interface* adalah *public* dan *abstract*.
Bisa mendeklarasikan *constant* dan *instance variable*.	Hanya bisa mendeklarasikan *constant*. Secara implisit variabel yang dideklarasikan di *interface* bersifat *public, static* dan *final*.
Method boleh bersifat *static*.	*Method* tidak boleh bersifat *static*.
Method boleh bersifat *final*.	*Method* tidak boleh bersifat *final*.
Suatu *abstact class* hanya bisa meng-*extend* satu *abstract class* lainnya.	Suatu *interface* bisa meng-*extend* satu atau lebih *interface* lainnya.
Abstract class hanya bisa meng-*extend* satu *abstract class* dan meng-*implement* beberapa *interface*.	Suatu *interface* hanya bisa meng-*extend interface* lainnya. Dan tidak bisa meng-*implement class* atau *interface* lainnya.

Sebagai contoh, kita akan mengubah *class* `Komponen.java` menjadi *interface*, lalu tambahkan beberapa *variable* dan *method*, seperti berikut ini:

Definisi Interface (komponen.java)

```
interface Komponen
{
    //Secara Implisit Variable tersebut Bersifat Public Static dan Final
    double kecepatan = 160.0;
    String model = "Lamborghini";

    //Secara Implisit Method Tersebut Bersifat Public dan Abstrak
    void mesin();
    void design();
}
```

Berbeda dengan *abstract class*, komponen didalam *interface* memiliki aturan dimana kita tidak perlu menambahkan sifat seperti *public*, *abstract* dan *final* pada *variable* atau *method* tersebut, karena secara implisit *variable* atau *method* tersebut sudah bersifat *public*, *abstract* dan *final*.

Buatlah class baru dan namakan `Mobil.java`. Pada *class* tersebut kita perlu mengimplementasikan *class* `Komponen.java` dengan menggunakan keyword *implements*.

Deinisi Class (mobil.java)

```
public class Mobil implements Komponen
{
    public static void main(String[] args)
    {
        Mobil data = new Mobil();
```

```
    }

    @Override
    public void mesin()
    {
    }

    @Override
    public void design()
    {
    }
}
```

Selanjutnya kita bisa melakukan apa saja pada method-method tersebut, sebagai contoh seperti berikut ini:

Program (Mobil.java)

```
public class Mobil implements Komponen
{
    public static void main(String[] args)
    {
        Mobil data = new Mobil();
        data.mesin();
        data.design();
    }

    @Override
    public void mesin()
    {
        System.out.println("Kecepatan Mobil: "+kecepatan);
    }

    @Override
    public void design()
    {
        System.out.println("Mempunyai Model: "+model);
    }
}
```

Output Program (mobil.java)

```
Kecepatan Mobil: 160.0
Mempunyai Model: Lamborghini
```

REFERENSI:

[1] Lewis, John, dan William Loftus. 2015. *Java Software Solutions Foundations of Program Design 8th Edition*. London: Pearson Education.

[2] Horstmann, Cay S. 2012. *Big Java: Late Objects, 1st Edition*. United States of America: John Wiley & Sons, Inc.

[3] Gaddis, Tony. 2016. *Starting Out with Java: From Control Structures through Objects (6th Edition)*. Boston: Pearson.